dhp Documentation

Release 0.0.14

Jeff Hinrichs

February 05, 2016

Contents

CHAPTER 1

Dirty Hungarian Phrasebook

dhp is a library of snippets, almost guaranteed to get you into trouble.

I obtained it, from a vendor, on the corner, outside of PyCon.

Actually, this is a growing repository of routines that I find helpful from time to time. I think you might too.

1.1 Phrasebook Examples

dhp.doq – Use ORM like expressions to query simple data sources.

dhp.search – Search related method and functions.

- *fuzzy_search* - search like "Sublime Text"

dhp.structures – Unique structures that build on Python's built-ins.

- **DictDot** - Ever wish the dictionary supported dot access?

dhp.test – Helpful test helper routines.

- *tempfile_containing* - generate a temporary file that contains indicated contents and returns the filename for use. When finished the tempfile is removed.

dhp.transforms

- *to_snake* - transform a "camelCased" name into a pythonized version, "camel_cased".

dhp.VI

- *iteritems* - return the proper iteritems method for a dictionary based on the version of Python

dhp.xml

- *xml_to_dict* - parse any ugly, but valid, xml to a python dictionary.
- *ppxml* - format/reformat any ugly but valid xml, a pretty printer for xml

1.2 Supports

Tested on Python 2.7, 3.2, 3.3, 3.4

1.3 Requirements

None.

1.4 Installation

Make sure to get the latest version.

```
pip install dhp
```

1.5 Download

- https://pypi.python.org/pypi/dhp

1.6 Project Site

- https://bitbucket.org/dundeemt/dhp

1.7 License

BSD

1.8 Documentation

- http://dhp.rtfd.org/

1.9 Change Log

See Change Log

1.10 Contributing

See Contributing

1.11 Contributors

See Contributors

Contents:

1.11.1 Change Log

0.0.14 (dev)

- added dhp.structures.ComparableMixin to aid in creating classes with rich comparisons.
- dropped support for Python3.2

0.0.13 (released 2015-12-20)

- integration with Appveyor CI for windows testing
- `dhp.tempus` - humane time interval transforms
  - dhp.tempus.interval_from_delta - transform a datetime.timedelta to an interval.
  - dhp.tempus.delta_from_interval - transform an interval string to a datetime.timedelta object.
- `dhp.cache` - a simple cache class

0.0.12 (released 2015-09-27)

- dhp.doq.DOQ - implemented range operator for lookups
- dhp.VI - StringIO - export StringIO from the proper package based on py2/py3
- dhp.search - Improved documentation.
- dhp.math - Improved documentation. Improved type tolerance. int/float/decimal
- test coverage improved on all submodule that were less than 100%

0.0.11 (released 2015-09-23)

- dhp.doq.DOQ - Duke is on the job to handle all your simple data source querying needs.

0.0.10 (released 2015-09-22)

- dhp.structures.DictDot - initial implementation

0.0.9 (released 2015-08-23)

- dhp.search.fuzzy_search - made case insensitive

0.0.8 (released 2015-08-19)

- refactor of test suite now that we are using *pip install -e .*

0.0.7 (released 2015-06-27)

- dhp.search.fuzzy_search and .fuzzy_distance

1.11.2 Contributing

Notes on how to contribute

Setting up a dev environment

These instructions assume you are developing in a virtualenv, you are, aren't you?

1. Clone the code into your virtualenv
2. You should have the packages in *dev-requirements.txt* installed

```
pip install -r requirements-dev.txt
```

3. install dhp as editable

```
pip install -e .
```

4. Tests should be passing locally

```
py.test
```

5. Editing documentation - you will need to build the docs initially then use docwatch, to auto build the docs when saved as you edit.

```
cd docs
make html
cd ..
```

```
python docwatch.py
```

Pull Requests

- Code should be passing all tests locally, bonus points for passing drone.io
- New code should have new tests to go along with it.
- Code should be pep8 compliant
- update documentation as necessary
- update contributors.rst
- make a pull request

1.11.3 Contributors

People who have contributed to the project

- Jeff Hinrichs <jefffh (at) dundeemt.com>

1.11.4 dhp.doq

DOQ

pronounced *Duke* allows you to query an list, iterable or generator of objects with a Django ORM like / Fluent interface. This is useful for exploratory programming and also it is just a nice, comfortable inteface to query your data

objects. DOQ supports lazy evaluations and nested objects.

Example

Say you had a csv file of employee records and you wanted to list the employees in the IT department. Well you could do the traditional thing or ...

```
EmployeeRecord = namedtuple('EmployeeRecord', 'emp_id, name, dept, hired')


def csvtuples():
    '''csv named tuple emitter.'''
    reader = csv.reader(TEST_FILE)
    for emp in map(EmployeeRecord._make, reader):
        yield emp

doq = DOQ(data_objects=csvtuples())
for emp in doq.filter(dept='IT'):
    print(emp)

# Now let's list everyone who is not in IT.
for emp in doq.exclude(dept='IT'):
    print(emp)

# ok, now let's sort the not IT employees by name
for emp in doq.exclude(dept='IT').order_by('name'):
    print(emp)
```

Yes, it is just that easy. You can chain .filter() and .exclude(). There is a .get method that raises DoesNotExist and MultipleObjectsReturned. All that ooohey gooey goodness of an full blown ORM but quick and easy and works without a lot of setup.

Let's throw some remote json data at the Duke and see what happens.

```
from dhp.structures import DictDot
from dhp.doq import DOQ
import requests

def json_ds(url):
    # fetch some json data, transform the returned dict to DictDot so
    # we can access attributes with dotted notation and then return
    # a DOQ with that data.
    data_objects = [DictDot(x) for x in requests.get(url).json()]
    return DOQ(data_objects=data_objects)

users = json_ds('http://jsonplaceholder.typicode.com/users')
type(users)         # prints  <class 'dhp.doq.DOQ'>
users.all().count   # prints 10
user = users.all()[0]
type(user)       # prints  <class 'dhp.structures.DictDot'>
user.id          # prints  1
user.address.suite  # prints  u'Apt. 556'
users.filter(address__suite__startswith='Apt.').count     # prints 3
```

One quick note before we head into the full documenation. DOQ is NOT a full blown Object Relation Manager. It does not create databases, nor know how to access them. If that is what you desire, then SQLAlchemy, Pony, PeeWeeDB or Django's ORM is probably going to get you what you want.

If you are looking to slap some lipstick on a simple data source, well then, DOQ is your color. dhp.doq package for api specifics.

1.11.5 dhp.math

fequal

compare to floats to see if they are equal within a tolerance

fequal(*num1*, *num2*, *tolerance=0.000001*)

return True if num1 is within tolerance of num2, else false

Parameters

- **num1** – float
- **num2** – float
- **tolerance** – float

Return type boolean

```
from dhp.math import fequal

assert fequal(1.123456, 1.1234561)
```

Use case: comparing floats can be interesting due to internal representations

is_even

returns True if integer is even

is_even(*num*)

Parameters **num** – int

Return type boolean

is_odd

returns True if integer is odd

is_odd(*num*)

Parameters **num** – int

Return type boolean

mean

returns the Arithmetic mean (a/k/a average) of a list of numbers

mean(*lst*)

Parameters **list** – float | int | mixed

Return type float

gmean

returns the Geometric mean of a list of numbers

gmean (*lst*)

Parameters **list** – float | int | mixed

Return type float

hmean

returns the Harmonic mean of a list of numbers

hmean (*lst*)

Parameters **list** – float | int | mixed

Return type float

1.11.6 dhp.search

fuzzy_search

given a list of strings(haystack) to search, return those elements, ranked, that fuzzily match the search term(needle).

fuzzy_search (*needle*, *haystack*)
return a ranked list of elements from haystack that fuzzily match needle

Parameters

- **needle** – what you are searching to find
- **haystack** – list of things to search

Return type ranked sublist of haystack elements matching needle

```
from dhp.search import fuzzy_search

haystack = ['.bob', 'bob.', 'bo.b', 'fred']
assert fuzzy_search(needle='bob', haystack) == ['bob.', '.bob', 'bo.b']
```

Use case: create a "Sublime Text" like search experience

1.11.7 dhp.structures

DictDot

DictDot subclasses Python's built-in dict object and offers attribute access to the dictionary. A little code says alot:

```
from dhp.structures import DictDot

my_dict = {'hovercraft': 'eels', 'speed': 42}
dicdot = DictDot(my_dict)
assert dicdot.hovercraft == 'eels'
assert dicdot.speed == 42

# ok, how about this?
```

```
dicdot = DictDot(hovercraft='eels', speed=42)
assert dicdot.hovercraft == 'eels'
assert dicdot.speed == 42

# or if your attacker has a pointed stick
dicdot = DictDot(my_dict, bunch='bananas')
assert dicdot.speed == 42
assert dicdot.bunch == 'bananas'

dicdot.new_value = 17
assert dicdot['new_value'] == 17
assert dicdot['hovercraft'] == 'eels'

# and now this ...
import json
assert json.dumps(dicdot) == '{"new_value": 17, "speed": 42, "hovercraft": "eels", "bunch": "bananas"
```

All of the methods and functions of a normal Python dictionary are present and available for you to use.

Use case: Those times when you don't want to type *{"..."}* but still want the goodness that is Python's dictionary.

ComparableMixin

To implment comparisions and sorting for your classes just subclass the mixin and then implement the _cmpkey() method:

```
from dhp.structures import ComparableMixin

class Comparable(ComparableMixin):
    def __init__(self, value):
        self.value = value

    def _cmpkey(self):
        return self.value
```

The magic methods lt, le, eq, ge, gt are all implemented and NotImplemented is returned when appropriate. Easier to use than `functools.total_ordering`. see https://wiki.python.org/moin/HowTo/Sorting for information on how the output of _cmpkey will sort.

1.11.8 dhp.tempus

This module includes tools to deal with time, dates, and intervals.

delta_from_interval

return a python datetime.timedelta that is repesented by an human parseable Interval string. NwNdNhNmNs, i.e. 1w2d3h4m5s - One week, 2 days, 3 hours, 4 minutes and 5 seconds. Which can be quite useful if you want a human to schedule a delay or time based repeat interval.

delta_from_interval (*interval*)
return a python datetime.timedelta represented by interval.

Parameters **interval** – str

Return type datetime.timedelta

```
from dhp.tempus import delta_from_interval

for k, val in iteritems(my_dict):
    do_something(k, val)
```

Use case: supporting python2 code that uses iteritems when targeting both 2 and 3.

PY_VER

is set to the major version of python currently running. Either 2 or 3 respectively.

StringIO

Imports the correct StringIO for the currently running version of Python.

```
from dhp.VI import StringIO
```

1.11.9 dhp.test

tempfile_containing

generate a temporary file that contains indicated contents and returns the filename for use. When finished the tempfile is removed.

tempfile_containing(*contents*[, *suffix=''*])

Generate a temporary file with contents specified, clean up when done.

Parameters

- **contents** – what should be written to the temp file
- **suffix** – *optional* suffix of temp file, if required

Return type filename as string

```
from dhp.test import tempfile_containing

contents = 'I will not buy this record, it is scratched.'
with tempfile_containing(contents) as fname:
    do_something(fname)
```

Use case: When testing, some functions/modules expect one or more file names to process. This phrase creates a temporary file via Python's `mkstemp`, writes the contents to it and closes the file so there is no contention with the module being tested on any platform. When the `with` statement goes out of scope, it cleans up the temporary file.

1.11.10 dhp.transforms

to_snake

given a "camelCase" string, transform it into a python-esque "camel_case".

to_snake(*name*)

return pythonized format of name, assumes name is some camelCase variant.

Parameters **name** – camel cased named to transform

Return type a pythonized string representation of the camel cased name.

```
from dhp.transforms import to_snake

assert to_snake('camelCase') == 'camel_case'
```

Use case: helpful when converting awful xml that uses camelCase to a python representation.

1.11.11 dhp.xml

xml_to_dict

There are a number of examples, on the intertubes, of doing this exact thing. However, many of them die on attributes. This has proven to be a robust routine and has dealt with all valid xml thrown at it.

xml_to_dict (*xml*)
convert valid xml to a python dictionary

Parameters **xml** – string containing xml to be converted

Return type dictionary

```
from dhp.xml import xml_to_dict

xml = '<vehicle type="Hovercraft"><filled/><cargo>eels</cargo></vehicle>'
xml_to_dict(xml)

{'vehicle': {'@type':'Hovercraft',
             'cargo':'eels',
             'filled': None}
}
```

Use case: parse any ugly, but valid, xml to a python dictionary.

ppxml

Pretty print xml. reformat xml in a sane way. Often times xml from external/3rd party sources is delivered like a gigantic furball, making it hard for a human to parse/read, this utility function makes it a bit more palatable.

ppxml (*xml*)
format xml for easier viewing

Parameters **xml** – string containing xml to be formatted

Return type string

```
>>> from dhp.xml import ppxml
>>> xml = '<vehicle type="Hovercraft"><filled/><cargo>eels</cargo></vehicle>'
>>> ppxml(xml)
u'<?xml version="1.0" ?>\n<vehicle type="Hovercraft">\n  <filled/>\n  <cargo>eels</cargo>\n</vehicle>
>>> print ppxml(xml)
<?xml version="1.0" ?>
<vehicle type="Hovercraft">
  <filled/>
  <cargo>eels</cargo>
</vehicle>
```

1.11.12 dhp.VI

These are simple methods for dealing with Python 2/3 compatibility issues. They are focused on solving the problems of python 2/3 support in the dhp package. If you need more see six

iteritems

return the proper iteritems method for a dictionary based on the version of Python

iteritems(*dct*)
return proper iteritems method

Parameters **dct** – dictionary

Return type iterable method

```
from dhp.VI import iteritems

for k, val in iteritems(my_dict):
    do_something(k, val)
```

Use case: supporting python2 code that uses iteritems when targeting both 2 and 3.

PY_VER

is set to the major version of python currently running. Either 2 or 3 respectively.

StringIO

Imports the correct StringIO for the currently running version of Python.

```
from dhp.VI import StringIO
```

1.11.13 Release Procedures

Notes on how to prepare, package and release a new version

Pre-Release

1. You should have the packages in *requirements-dev.txt* installed

```
pip install -U -r requirements-dev.txt
```

2. Code should be checked in

```
hg sum --remote
```

3. Tests should be passing locally

```
py.test -v
```

4. drone tests should be passing – https://drone.io/bitbucket.org/dundeemt/dhp/latest
5. Update the changelog
6. Read the Docs builds should be building cleanly – http://dhp.readthedocs.org/en/latest/

7. Run the release script in –dry-run mode and check that no errors or issues are outstanding. Specifically, check version information from bumpversion.

```
./release.sh --dry-run
```

Release

bumping the version, checking the build, committing tags

1. Run the release script

```
./release.sh
```

2. Push the commit

```
hg push
```

3. Verify drone builds – https://drone.io/bitbucket.org/dundeemt/dhp/latest
4. Verify docs built – http://dhp.readthedocs.org/en/latest/
5. Set the default docs to the new version – https://readthedocs.org/dashboard/dhp/versions/
6. upload to pypi

```
twine upload dist/dhp-x.y.z.tar.gz
```

7. InsecurePlatformWarning - If you get this warning on python2.7+ you will need to install some additional modules

```
pip install pyopenssl ndg-httpsclient pyasn1
```

8. Check PyPi for problems and make sure docs and package is correct – https://pypi.python.org/pypi/dhp

Profit

You and the rest of the world can enjoy

1.11.14 API Documentation

dhp package

Subpackages

dhp.VI package

Module contents collection of routines to support python 2&3 code in this package

dhp.VI.**iteritems**(*dct*)
return the appropriate method

dhp.VI.**py_ver**()
return the Major python version, 2 or 3

Exports The following are exported by dhp.VI

StringIO The proper version of StringIO from cStringIO or io package.

```
from dhp.VI import StringIO
```

dhp.doq package

Module contents Data Object Query mapper.

pronounced *Duke* allows you to query an list, iterable or generator yielding objects with a Django ORM like / Fluent interface. This is useful for exploratory programming and also it is just a nice, comfortable inteface to query your data objects.

Example

Say you had a csv file of employee records and you wanted to list the employees in the IT department. Well you could do the traditional thing or ...

Example:

```
# bread and butter Python
EmployeeRecord = namedtuple('EmployeeRecord', 'emp_id, name, dept, hired')

def csvtuples():
    '''csv named tuple generator.'''
    reader = csv.reader(TEST_FILE)
    for emp in map(EmployeeRecord._make, reader):
        yield emp

# Enter the Duke
doq = DOQ(data_objects=csvtuples())
for emp in doq.filter(dept='IT'):
    print(emp)

# Now let's list everyone who is not in IT.
for emp in doq.exclude(dept='IT'):
    print(emp)

# ok, now let's sort the not IT employees by name
for emp in doq.exclude(dept='IT').order_by('name'):
    print(emp)
```

Yes, it is just that easy. You can chain *filter()* and *exclude()*. There is a *get()* method that raises *DoesNotExist()* and *MultipleObjectsReturned()*.

All that ooohey gooey query goodness of a traditional ORM but quick and easy and works without a lot of setup.

One quick note before we head into the full documenation. DOQ is NOT a full blown Object Relation Manager. It does not create databases, nor know how to access them. If that is what you desire, then SQLAlchemy, Pony, PeeWeeDB or Django's ORM is probably going to get you what you want.

If you are looking to slap some lipstick on a simple data source, well then, DOQ is just your color.

class dhp.doq.**DOQ**(*data_objects*)

Bases: object

data object query mapper.

all()
Returns a cloned DOQ. Short hand for an empty filter but it reads more naturally than `doq.filter()`.

Parameters **None** –

Returns A cloned DOQ object.

Return type *DOQ*

Example:

```
for obj in doq.all():
    print(obj)
```

count
A property that returns the number of objects currently selected. Can also use len(doq).

Returns The number of objects selected.

Return type (int)

Example:

```
if doq.filter(name='Jeff').count == 1:
    do_something
result = doq.filter(emp_id=1)
assert doq.count == len(doq)
```

exclude(***look_ups*)
Returns a new DOQ containing objects that **do not match** the given lookup parameters.

Parameters **look_ups** – The lookup parameters should be in the format described in *Attribute Lookups* below. Multiple parameters are joined via AND in the underlying logic, and the whole thing is enclosed in a NOT.

Returns A cloned DOQ object with the specified exclude(s).

Return type *DOQ*

Raises `AttributeError` – If an attribute_name in the look_ups specified can not be found.

This example excludes all entries whose hired date is later than 2005-1-3 AND whose name is "Jeff":

```
doq.exclude(hired__gt=datetime.date(2005, 1, 3), name='Jeff')
```

filter(***look_ups*)
Returns a new DOQ containing objects that match the given lookup parameters.

Parameters **look_ups** – The lookup parameters should be in the format described in *Attribute Lookups* below. Multiple parameters are joined via AND in the underlying logic.

Returns A cloned DOQ object with the specified filter(s).

Return type *DOQ*

Raises `AttributeError` – If an attribute_name in the look_ups specified can not be found.

Example:

```
doq.filter(name='Foo', hired__gte='2012-01-03')
```

get(***look_ups*)
Preform a get operation using 0 or more filter keyword arguments. A single object should be returned.

Parameters **look_ups** – The lookup parameters should be in the format described in *Attribute Lookups* below. Multiple parameters are joined via AND in the underlying logic.

Returns A single matching data_object from data_objects.

Return type data_object

Raises `AttributeError` – If an attribute_name in the look_ups specified can not be found.

Example:

```
obj = doq.get(emp_id=1)
```

Raises

- *DoesNotExist* – If no matching object is found.
- *MultipleObjectsReturned* – If more than 1 object is found.

static get_attr(*obj*, *attrname*)

Retrieve a possibly nested attribute value.

Parameters

- **obj** (*data object*) – The data object to retrieve the value.
- **attrname** (*str*) – The attribute name/path to retrieve. A simple object access might be *name*, a nested object value might be *address__city*

Returns The value of the indicated attribute.

order_by(**attribute_names*)

Return a new DOQ with thes results ordered by the data_object's attribute(s). The default order is assending. Use a minus (-) sign in front of the attribute name to indicate descending order. Repeated .order_by calls are NOT additive, they replace any existing ordering.

Parameters **attribute_names** – 0 or more data_object attribute names. Listed from most significant order to least.

Returns A new DOQ object with the specified ordering.

Return type *DOQ*

Example:

```
doq.all().order_by('emp_id')  # emp_id 1, 2, 3, ..., n
doq.all().order_by('-emp_id') # emp_id n, n-1, n-2, ..., 1

doq.all().order_by('dept', 'emp_id') # by dept, then by emp_id
```

to order randomly, use a '?'.

```
doq.all().order_by('?')
```

static order_by_key_fn(*attrname*)

Override this method to supply a new key function for the order_by method.

The default function is:

```
lambda obj: DOQ.get_attr(obj, attrname)
```

If you had an attribute "emp_id" that returned a number as a string `['2', '1', '3', '11']`. It would be ordered by string conventions returning them in `['1', '11', '2', '3']`. If you want them sorted like integers `['1', '2', '3', '11']`, you would subclass DOQ and override the `order_by_key_fn` like this:

```
        class MyDOQ(DOQ):
            @staticmethod
            def order_by_key_fn(attrname):
                if attrname == 'emp_id':
                    def key_fn(obj):
                        # return attr as an integer
                        return int(DOQ.get_attr(obj, attrname))
                else:
                    def key_fn(obj):
                        # return the standard function.
                        return DOQ.get_attr(obj, attrname)
                return key_fn

        mydoq = MyDOQ(data_objects)
        mydoq.all().order_by('emp_id')
```

Parameters **attrname** (*str*) – The attribute name be acted on by the order_by method.

Returns

A function that takes the attribute name as an argument and that also has access to the object be acted on.

Return type function

Raises AttributeError – If the attribute_name specified can not be found.

ordered

True if an order is set, otherwise False.

Returns True if the order_by is set, otherwise False.

Return type bool

Example:

```
        results = doq.all()
        assert results.ordered == False
        results = results.order_by('name')
        assert results.ordered == True
```

exception dhp.doq.**DoesNotExist**

Bases: exceptions.Exception

Raised when no object is found.

exception dhp.doq.**MultipleObjectsReturned**

Bases: exceptions.Exception

raised when more than 1 object returned but should not be.

Attribute Lookups Attribute lookups are similar to how you specify the meat of an SQL WHERE clause. They're specified as keyword arguments to the DOQ methods *filter()*, *exclude()* and *get()*.

The format of look_ups is attribute_name__operation=value That is the name of the attribute to look at, a double under score(dunder) and then the lookup operator, an equals sign and then the value to compare against. The format was inspired by Django's ORM.

DOQ's inbuilt lookups are listed below.

As a convenience when no lookup type is provided (like in `doq.get(emp_id=14)`) the lookup type is assumed to be *exact*.

exact Exact case-sensitive match.

```
doq.get(emp_id__exact=4)
assert doq.get(name='Jeff') == doq.get(name__exact='Jeff')
```

iexact Exact, case insensitive, match.

```
doq.filter(name__iexact='jeff')  # would match, jEFF, Jeff, etc.
```

lt Less Than.

```
doq.filter(emp_id__lt=3)         # given [4, 3, 2, 1], would match [2, 1]
```

lte Less Than or Equal to.

```
doq.filter(emp_id__lte=3)         # given [4, 3, 2, 1], would match [3, 2, 1]
```

gt Greater Than.

```
doq.filter(emp_id__gt=3)         # given [4, 3, 2, 1], would match [4, ]
```

gte Greater Than or Equal To.

```
doq.filter(emp_id__gte=3)         # given [4, 3, 2, 1], would match [4, 3]
```

contains If the value is in the attribute.

```
doq.filter(name__contains='o') # given ['Oscar', 'John', 'Jo'], would match ['John', 'Joe']
```

icontains Case insensitive version of contains. See above.

```
doq.filter(name__icontains='o') # given ['Oscar', 'John', 'Jo'], would match ['Oscar', John', 'Joe']
```

startswith If the attribute value startswith.

```
doq.filter(name__startswith='O') # given ['Oscar', 'John', 'Jo'], would match ['Oscar', ]
```

istartswith Case insensitive version of startswith. See above.

```
doq.filter(name__istartswith='o') # given ['Oscar', 'John', 'Jo'], would match ['Oscar', ]
```

endswith If the attribute value endswith.

```
doq.filter(name__endswith='n') # given ['Oscar', 'John', 'Jo'], would match ['John', ]
```

iendswith Case insensitive version of endswith. See above.

```
doq.filter(name__iendswith='N') # given ['Oscar', 'John', 'Jo'], would match ['John', ]
```

in If the attribute value is in the list supplied.

```
doq.filter(emp_id__in=[1, 3])    # given [1, 2, 3, 4], would match [1, 3]
```

range Is a short hand equivalent of a >= b and a <= c where *a__range=(b, c)* and b <= c

```
doq.filter(emp_id__range=(2, 5))  # is equivalent of doq.filter(emp_id__gte=2, emp_id__lte=5)
```

Nested Objects If you have a object that is composed of nested objects, you can access the values of the nested subobjects by using double underscores to list the path of the relationship. Say you had a list of objects with the following layout:

```
user:
    id
    name
    address:
        street
        suite
        zipcode
        geo:
            lat
            lon
```

You would access the top-level attributes.

```
doq.filter(id=7)`
```

To access the suite information,

```
doq.filter(address__suite='Apt. 201')
```

which would be an exact match on the attribute value. To use another operator with your lookup just specify it.

```
doq.filter(address__suite__startswith='Apt.')
```

Ordering on a nested attribute is the same. To order by lat:

```
doq.all().order_by('address__geo__lat')
```

Slicing DOQ (Limiting) Slicing a DOQ is supported. Since we are not performing SQL the results of a slicing operation are immediate and return a list of data_objects.

```
>>> type(doq.all()[2:4])
<type 'list'>
```

This also means that Negative indexing is supported.

```
doq.all()[-1]
```

Would return the last data_object from the results.

dhp.math package

Module contents handy math and statistics routines

Supported Number sets

- {int} = Set of integers
- {float} = Set of float
- {decimal} = Set of Decimal
- {mixed-float} = {float} + {int}
- {mixed-decimal} = {decimal} + {int}

Return Type Precedence The type returned is based on the function, input type(s), The simplest meaningful type is returned.

- bool
- int
- float
- Decimal

exception dhp.math.**MathError**
Bases: exceptions.ValueError

general math error

exception dhp.math.**UndefinedError**
Bases: *dhp.math.MathError*

When the calculation is undefined for the given input.

dhp.math.**fequal**(*num1*, *num2*, *delta=1e-06*)
Compare equivalency of two numbers to a given delta.

Both num1 and num2 must be from the same set of {mixed-float} OR {mixed-decimal}.

$num1 \equiv num2 \iff |num1 - num2| < delta$

Parameters

- **num1** (*{mixed-float} | {mixed-decimal}*) – The first number to compare.
- **num2** (*num1*) – The second number to compare.
- **delta** (*float*) – The amount of difference allowed for equivalence. (default: 0.000001)

Returns

True if the absolute difference between *num1* and *num2* is less than *delta*, else False.

Return type (bool)

Raises TypeError – If testing a float and a Decimal.

dhp.math.**gmean**(*nums*)
Return the geometric mean of the list of numbers.

$G = (x_1 * x_2 * ... * x_N)^{\frac{1}{N}} = (\prod_{i=1}^{N} x_i)^{\frac{1}{N}}$

Parameters **nums** (*list*) – list of numbers ({mixed-float}|{mixed-decimal})

Returns Geometric Mean of the list.

Return type (float|decimal)

Raises

- (UndefinedError) – If nums is empty. $N = 0$
- (TypeError) – If nums contains both float and Decimal numbers.

dhp.math.**hmean**(*nums*)

Return the harmonic mean of a list of numbers.

$$H = \frac{N}{\frac{1}{x_1} + \frac{1}{x_1} + ... + \frac{1}{x_N}} = \frac{N}{\sum_{i=1}^{N} \frac{1}{x_i}}$$

Parameters **nums** (*list*) – list of numbers ({mixed-float}|{mixed-decimal})

Returns Harmonic Mean of the list.

Return type (float|decimal)

Raises (UndefinedError) – If the list is empty. $N = 0$

dhp.math.**is_even**(*num*)

Return True if num is even, else False.

An integer is even if it is 'evenly divisible' by two.

$Even = \{2k : k \in Z\}$

Parameters **num** (*int*) – The num to check.

Returns True if num is even, else False.

Return type (bool)

Raises (MathError) – If num is not an integer.

dhp.math.**is_odd**(*num*)

rReturn True if num is odd, else False.

An integer is odd if it is not even.

$Odd = \{2k + 1 : k \in Z\}$

A number expressed in the binary is odd if its last digit is 1 and even if its last digit is 0.

Parameters **num** (*int*) – The num to check.

Returns True if num is odd, else False.

Return type (bool)

Raises (MathError) – If num is not an integer.

dhp.math.**mean**(*nums*)

Return the arithmetic mean of the list of numbers

$$\bar{X} = \frac{x_1 + x_2 + ... + x_N}{N} = \frac{\sum_{i=1}^{N} x_i}{N}$$

Parameters **nums** (*list*) – list of numbers ({mixed-float}|{mixed-Decimal})

Returns Arithmetic Mean of the list.

Return type (float|decimal)

Raises

- (UndefinedError) – If nums is empty. $N = 0$

- (TypeError) – If nums contains both float and Decimal numbers.

dhp.math.**median**(*nums*)
Return the median value from the list.

Given: $a < b < c < d$ The median of the list [a, b, c] is b, and, the median of the list [a, b, c, d] is the mean of b and c; i.e. $\frac{b+c}{2}$

Parameters **nums** (*list*) – list of numbers ({mixed-float}|{mixed-decimal})

Returns The median of the list of numbers.

Return type (int|float|decimal)

dhp.math.**mode**(*lst*)
Return the mode (most common element value) from the list.

Parameters **lst** (*list*) – list of hashable objects to search for the mode.

Returns The most common value in lst.

Return type (list element)

Raises

- (UndefinedError) – If lst is empty.
- (MathError) – If lst is multi-modal.

dhp.math.**pstddev**(*lst*)
return the population standard deviation of the elements in the list

dhp.math.**pvariance**(*lst*)
return the population variance for the list of numbers

dhp.math.**sstddev**(*lst*)
return the sample standard deviation of the elements in the list

dhp.math.**svariance**(*lst*)
return the sample population variance for the list of numbers

dhp.math.**ttest_independent**(*lst1*, *lst2*)
calc the ttest for two independent samples

dhp.search package

Module contents search type utilities

dhp.search.**fuzzy_distance**(*needle*, *straw*)
calculate distance between needle and a straw from the haystack.

Parameters

- **needle** (*str*) – The thing to match
- **straw** (*str*) – The thing to match against

Returns A distance of 0 indicates a search failure on one or more chars in needle. The lower the distance the closer the match, matching earlier and closer together results in a shorter distance.

Return type (int)

dhp.search.**fuzzy_search**(*needle*, *haystack*)
Return a list of elements from haystack, ranked by distance from needle.

Parameters

- **needle** (*str*) – The thing to match.
- **haystack** (*list*) – A list of strings to match against.

Returns

Of strings, ranked by distance, that fuzzy match needle to one degree or another.

Return type (list)

Example:

```
corpus = ['django_migrations.py',
          'django_admin_log.py',
          'main_generator.py',
          'migrations.py',
          'api_user.doc',
          'user_group.doc',
          'accounts.txt',
          ]
assert fuzzy_search('mig', corpus) == ['migrations.py',
                                       'django_migrations.py',
                                       'main_generator.py',
                                       'django_admin_log.py']
```

dhp.structures package

Module contents dhp data structures

class dhp.structures.**ComparableMixin**

Bases: object

Mixin to give proper comparisons.

Example:

```
class Comparable(ComparableMixin):
    def __init__(self, value):
        self.value = value

    def _cmpkey(self):
        return self.value
```

Returns NotImplemented if the object being compared doesn't support the comparison.

Raises NotImplementedError if you have not overridden the _cmpkey method.

Code is from Lennart Regebro https://regebro.wordpress.com/2010/12/13/python-implementing-rich-comparison-the-correct-way/

class dhp.structures.**DictDot**(**args*, ***kwargs*)

Bases: dict

A subclass of Python's dictionary that provides dot-style access.

Nested dictionaries are recursively converted to DictDot. There are a number of similar libraries on PyPI. However, I feel this one does just enough to make things work as expected without trying to do too much.

Example:

```
    dicdot = DictDot({
        'foo': {
            'bar': {
                'baz': 'hovercraft',
                'x': 'eels'
            }
        }
    })
    assert dicdot.foo.bar.baz == 'hovercraft'
    assert dicdot['foo'].bar.x == 'eels'
    assert dicdot.foo['bar'].baz == 'hovercraft'
    dicdot.bouncy = 'bouncy'
    assert dictdot['bouncy'] == 'bouncy'
```

DictDot raises an AttributeError when you try to read a non-existing attribute while also allowing you to create new key/value pairs using dot notation.

DictDot also supports keyword arguments on instantiation and is built to be subclass'able.

dhp.test package

Module contents routines and snippets generally userful for testing

dhp.test.**tempfile_containing**(**args*, ***kwds*)
create a temporary file, with optional suffix and return the filename, cleanup when finished

dhp.transforms package

Module contents dhp transforms library

dhp.transforms.**to_snake**(*buf*)
pythonize the name contained in buf

dhp.xml package

Module contents routines generally helpful for dealing with icky xml

dhp.xml.**etree_to_dict**(*tree*)
transform element tree to a dictionary

dhp.xml.**ppxml**(*xmls*)
pretty print xml, stripping an existing formatting

dhp.xml.**xml_to_dict**(*xml_buf*)
convert xml string to a dictionary, not always pretty, but reliable

Module contents

dhp top level

Indices and tables

- genindex
- modindex
- search

d

U

X